

Supermon: A high-speed cluster monitoring system*

Matthew J. Sottile Ronald G. Minnich

Los Alamos National Laboratory[†]
Advanced Computing Laboratory
MS-B287
Los Alamos, NM 87545 USA
{matt,rminnich}@lanl.gov

Abstract

Supermon is a flexible set of tools for high speed, scalable cluster monitoring. Node behavior can be monitored much faster than with other commonly used methods (e.g., rstatd). In addition, Supermon uses a data protocol based on symbolic expressions (S-expressions) at all levels of Supermon, from individual nodes to entire clusters. This contributes to Supermon's scalability and allows it to function in a heterogeneous environment. This paper presents the Supermon architecture and discuss initial performance measurements on a cluster of heterogeneous Alpha-processor based nodes.

1 Introduction

This paper describes Supermon, a set of tools for high-speed, low-impact monitoring of terascale clusters. The Supermon tools gather data from individual compute nodes, combine data from all nodes into a single cluster image, and pass the data to clients making it naturally hierarchical. Supermon proves that low-impact monitoring can be achieved for higher than expected sampling rates (up to 6600 samples per second for data extraction from the OS), allowing data to be sampled at a time-scale small enough to observe characteristics of cluster behavior not previously visible.

Monitoring is the act of observing a system via a set of *sensors*. Monitoring can be either *periodic* or *reac-*

tive, among other choices; in a periodic system, samples of the sensor output are taken at regular intervals and used to determine the state of the system; in a reactive system an external event (perceived failure) causes activation of sensor reading as an aid to diagnosis.

Monitoring can be used as an aid in *management*. The values read from sensors can be used to make an informed decision about what actions to take. Once the actions are taken, the sensors can be read again to determine if the action improved the state of the system or if other actions need to be taken.

The state of the art of cluster node management is extraordinarily primitive. The single most-used monitoring tool is the “ping” command: the command sends a simple packet to a remote node, and the remote node is supposed to reply. Typically, if a ping command fails, the only option is to reboot the node, via reset or power off. Once the node is rebooted, any information about the problem that precipitated the failure is lost.

Even if the ping command succeeds, the node may still be unusable due to other problems. A commonly-used problem determination sequence is as follows:

- A user notices that a job has not completed in a normal way, and contacts the system administrator.
- The sysadmin tries to log onto the node, which may or may not succeed.
- If the login fails, the sysadmin pings the node. Even if the ping works, there is no way to get onto the node, so the sysadmin must walk to the node and reboot it.
- If the ping fails, then the only option is to walk to the node and reboot it.

*This research is funded by the Department of Energy's Office of Science.

[†]Los Alamos National Laboratory is operated by the University of California for the National Nuclear Security Administration of the United States Department of Energy under contract W-7405-ENG-36, LA-UR No. 02-3585.

As this discussion shows, there are only two sensors in use in most clusters: the first sensor is the process that supports remote login; the second sensor is the process that responds to remote pings. These sensors are almost always used in a reactive manner, i.e., in response to perceived problems, and in many cases long after the failure that caused the error. Actually isolating the failure takes second place to getting the node back up and running again; the result is a lack of long-term data to allow precise problem determination.

For terascale and beyond clusters these management techniques are not sufficient and in fact may be impossible. We need to move to periodic monitoring of cluster nodes, and we need to make much better use of the available data.

In addition, for production clusters where downtime is to be minimized and long runtimes for jobs are highly desirable, monitoring becomes an important practice required to accomplish these goals. Monitoring any system, from physical entities to the state of a computer, introduces a small but unavoidable perturbation into the actual system. In the case of cluster monitoring, the act of monitoring can cause additional load on both compute nodes and the interconnection network binding them together.

In this paper we describe Supermon, a high-speed, low-impact monitoring tool for terascale clusters. Supermon uses symbolic expressions to represent data at all levels (kernel, node, concentrator) making it naturally composable and hierarchical. We demonstrate the power of symbolic data representation for kernel data, both as an efficient mechanism for encapsulating complex datatypes, but as a self-describing format that can be applied to many problems. We will present performance results based on benchmarks using high sampling rates at the kernel, single-node, and whole cluster level. For the entire paper, our working environment will be the LANL ACL 128 node Alpha Linux cluster, composed of a mixed set of nodes including Compaq DS-10, API CS-20, and Compaq ES-40 systems containing 1, 2, and 4 CPUs respectively.

2 Related Work

As mentioned in the previous section, modern cluster monitoring only exists as an undisciplined set of steps that is performed as a *reaction* to failure. Products that claim to provide monitoring simply provide a GUI on top of a tool that performs a subset of these steps (i.e., ping) [4, 8].

Current Linux-based monitoring tools are based on the SunRPC “remote status” (rstat) protocol [7]. In previous work, we showed that the Linux implemen-

tation of rstatd is very slow and has severe impact on the machine being monitored [5]. Moreover, this protocol was defined almost 20 years ago, before systems with SMPs and hot-plug network interfaces and disks were common. The type of data it returns is fixed, and the quantities are fixed. For example, CPU and network interface information is gathered into a single field so there is no clean way to handle SMPs or machines with multiple network interfaces. The interface is not extensible, so there is no provision for new sensor data such as from hardware monitors.

In the same paper, to overcome the performance limitations of rstatd, we created a monitoring package (also called Supermon) which included a kernel patch allowing data to be extracted from the kernel using a call to `sysctl()`. The kernel module gathered all of the required data that rstat provided, and pushed them out through a single system call to the calling program. This increased performance significantly, allowing sampling rates of over 30Hz per node.

This paper addresses the other serious problem of rstat, the archaic, protocol interface that is not extensible and not composable.

3 Architecture

The Supermon cluster monitoring system is split into three distinct components as shown in Figure 1: a loadable kernel module providing data, a single node data server (mon), and a data concentrator (Supermon) to compose samples from many nodes into a single data sample. The kernel module provides data samples through an entry in the `/proc` filesystem on Linux, while the single node server and data concentrator allow clients to retrieve data samples through TCP sockets. Clients that wish to use this data must connect to one of the three components, all of which speak the same protocol, and parse the data into whatever form they require. This protocol is based on symbolic expressions, or s-expressions, originally introduced as part of the LISP programming language by John McCarthy in the 1950s as a recursively defined, simple symbolic format for data representation [3].

3.1 The Protocol

The Supermon protocol is a client-server protocol. The protocol packets consist of s-expressions. Unlike SunRPC packets, the s-expressions contain self-describing data, so none of the RPC Compiler or XDR [6] tools are needed. S-expressions were introduced as part of the LISP programming language in

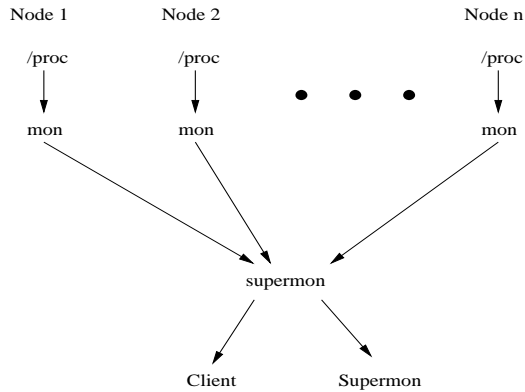


Figure 1. The major architectural components of Supermon and their relationships.

the 1950s. Their simple, recursive form allows them to encode arbitrarily complex data.

Another useful property of s-expressions is that they are not fixed-size, and they are not binary data. Standard RPC packets have a very strictly controlled binary format and size, as one way to achieve architecture independence. S-expressions in contrast can vary in size as needed, and achieve architecture independence by eliminating binary data entirely. As we discussed above, moving textual data instead of binary data has actually proven to be more efficient for Supermon.

We put the power of s-expressions to good use in Supermon. Possibly the most interesting aspect of the Supermon protocol is that it is *composable*. Supermon clients can serve as Supermon servers. An individual Supermon server can act as a client and aggregate the s-expression streams from multiple Supermon servers, and these servers in turn can also aggregate other streams. Standard RPC servers, such as automounters, can do limited composition, but aggregation is extremely difficult and involves loss of information from the initial server to the final client. The Supermon protocol supports aggregation with no loss of information. Scalability and composability are also important requirements for the protocol since it will be used on very large clusters. S-expressions are used at all levels of Supermon from the kernel module that provides the initial data to client applications making it naturally hierarchical. Each part of Supermon that requires s-expression manipulation is able to use a compact, efficient s-expression parser provided as part of the distribution.

```

(cpuinfo
  (user 232007070)
  (nice 1314934)
  (system 0))
(avenrun (avenrun0 2060) (avenrun1 2056)
  (avenrun2 2048))
(paging (pgpgin 16) (pgpgout 0) (pswpin 0)
  (pswpout 0))
(switch (switch 549615))
(time (timestamp 0xec05d78898)
  (jiffies 0xebd2e4b))
(netinfo
  (name lo eth0 eth1)
  (rxbytes 0 0 45681848671)
  (rxpackets 0 0 31522281)
  (rxerrs 0 3 0)
  (rxdrop 0 0 0)
  ...
)
  
```

Figure 2. Supermon output from the kernel module for the S command.

3.2 The Kernel Module

The kernel module is a dynamically loaded module that inserts an entry into the `/proc/sys` tree in the Linux kernel. The entry is a directory named **supermon** with two nodes, **S** and **#**.

The **S** entry returns Supermon data in s-expression form as shown in Figure 2. Data are grouped into categories. In each category are a number of named fields with their values. There can be more than one set of values for named fields, as can be seen in the **netinfo** entry. This machine has three ethernet interfaces (**lo**, **eth0**, and **eth1**) so each of the named fields has three elements, one for each interface. The first component of the **netinfo** list is the set of names of the interfaces. On a laptop, interfaces come and go as cards are plugged in. Supermon data will reflect this change: as the interfaces appear and disappear the size of the lists will change. Supermon data can be dynamic in a way that is difficult or impossible for **sysctl** entries.

The **#** entry returns data descriptors, also in s-expression format, as shown in Figure 3. The format of the lists is the same for all categories: category name (e.g., **cpuinfo**), cardinality of the category ((**nr 1**)), and the field names for the category (e.g., **user**). In the example shown, the **cpuinfo** field has a cardinality of one; on an SMP with 2 or 4 CPUs, it has a cardinality of 2 or 4. The **netinfo** category has a cardinality

```

(cpuinfo (nr 1) (user nice system)
)
(avenrun (nr 1) (avenrun0 avenrun1
                avenrun2))
(paging (nr 1) (pgpgin pgpgout pswpin
                pswpout))
(switch (nr 1) (switch))
(time (nr 1) (timestamp jiffies))
(netinfo (nr 3) (
  name rxbytes rxpackets rxerrs rxdrop
  rxfifo rxframe rxcompressed rxmulticast
  txbytes txpackets txerrs txdrop txfifo
  txcolls txcarrier txcompressed)
)

```

Figure 3. Supermon output from the kernel module for the # command.

of 3, since this machine has three interfaces; again, if interfaces come and go the cardinality will change.

Note that the data has structure that is self-describing and easily parsed by programs (or people). This output format is far more useful than the standard Linux `/proc` format entries, which we show in Figure 4. The standard Linux entries follow no particular format. `Cpuinfo` is presented as name-value pairs separated by colons. `Meminfo` is partly a table (first three lines) and partly name-value pairs (the rest of the lines). `Slabinfo` is the most confusing: at least the first line does describe the origin of the data i.e. the slab allocator, but the remaining lines present data which has no clear meaning.

We do not currently provide `swapinfo` or `meminfo` statistics. These have proven far too costly to query at speed. We described the problems in previous work [5].

3.3 Mon and Supermon

To move data from the kernel out to clients, two small server programs are required to provide the data via TCP at the single and multiple node level. At a single node, the kernel module provides data in its two entries in `/proc`. The “mon” server acts as the intermediate filter between `/proc` and TCP clients. It parses the s-expressions found in `/proc`, adding a minimal amount of information, and passes the data to clients on demand. For each client that connects, mon maintains a bitmask reflecting the fields of data that the particular client wants returned in a sample. This allows mon to filter data and reduce wasteful network traffic.

processor : 0
vendor_id : GenuineIntel
cpu family : 6
model : 8
model name : Pentium III (Coppermine)
...
total: used: free: shared: buffers: cached:
Mem: 262381568 233357312 29024256
0 25935872 78348288
Swap: 479473664 569344 478904320
MemTotal: 256232 kB
MemFree: 28344 kB
MemShared: 0 kB
...
slabinfo - version: 1.1 (statistics)
kmem_cache 54 58 136 2 2 1 : 54 54 2 0 0
...

Figure 4. Three different `/proc` entries: `cpuinfo`, `meminfo`, and `slabinfo`.

If clients wish to see a snapshot of a set of nodes in each sample, a second server called “Supermon” is provided. Supermon connects to a set of nodes that are running mon servers and acts as a data concentrator, presenting data sampled from many mon servers as a single data sample. The data format provided to clients by Supermon is identical to the format provided by mon. This allows many Supermon servers to be created, each sampling from a subset of the nodes within a cluster. Another Supermon could then be started to connect to the other Supermon servers monitoring portions of the cluster. Hierarchical Supermon servers can improve performance in situations where a cluster has many nodes and sampling rates are high. In addition to the ability to build hierarchical Supermons, Supermon also provides a similar bitmask based filter for each client, which is then used to improve efficiency between the Supermon/mon and Supermon/client connections.

4 Performance

The majority of time spent in development was dedicated to making each portion of the system as efficient as possible. Efficiency at each level allows Supermon to achieve the high sampling-rate goal that was unreachable with older monitoring tools. In this section, we present performance results for the kernel module, the kernel module plus mon, and the kernel module plus mon plus Supermon in multiple configurations.

4.1 Perturbation and monitoring

Before presenting the performance results that we have observed for the Supermon system, we should briefly discuss the reason why high peak sampling rates are critical for a good monitor. It is a well known fact that any monitoring system will perturb the system being monitored. Roughly speaking, perturbation in our context results from consumption of the monitored resources by the monitor itself. Any observation of a portion of the system, such as network traffic or CPU cycles, will reflect usage by both the monitor and the applications running on the system. Specifically, a particular metric will have an actual value S_{actual} and an observed value $S_{observed}$. Perturbation causes $S_{observed} = S_{actual} + S_{error}$, where S_{error} is the overhead caused by the monitor. Exact values are rarely necessary when monitoring, so some error ϵ in measurements is tolerable.

A good monitoring system will ensure that the overhead is lower than the tolerable error, so that $\epsilon > S_{error}$. Looking specifically at the relation of the peak sampling rate and this tolerable error, must consider what the peak sampling rate implies about the system. The sequence of software and hardware components involved in monitoring will have some bottleneck resource that will become saturated and act as a limiting factor for the peak sampling rate[2]. When this rate is reached, this component will be completely saturated. The monitor that can sample at a higher peak rate will not saturate this resource when used at the peak rate of the slower monitor.

Furthermore, if we assume that an application will require a fixed sampling rate significantly lower than either peak sampling rate, we know that the bottleneck resource will have a lower utilization for the system with a higher peak. This will result in a lower S_{error} term for the faster monitoring system, thus decreasing the perturbation caused by that monitor in sampling. If two monitoring systems are available with peak sampling rates such as R and $2R$, then the utilization of the saturated resource will be roughly $\frac{x}{R}$ and $\frac{x}{2R}$ respectively for a fixed sampling rate of x .¹ Since this resource utilization is intimately tied to the perturbation of the system by the monitor, it is clear why higher peak rates will yield a less intrusive monitor in practice.

¹Not all resource consumption is linearly related to the sampling rate, such as memory usage. It is the resources that are linearly related to the sampling rate that are important to consider at high sampling rates.

<i>Node Type</i>	<i>CPUs</i>	<i>Memory</i>	<i>Number</i>
DS-10	1	1GB	104
CS-20	2	2GB	16
ES-40	4	16GB	4
	152	200GB	124

Table 1. The LANL ACL xed testbed cluster.

<i>Benchmark</i>	<i>Data Size</i>	<i>Nodes</i>
Kernel	800 bytes	1
Mon	950 bytes	1
Supermon	(950*N) bytes	N

Table 2. Approximate data sizes for each benchmark.

4.2 The benchmark environment

The environment in which we ran all of our benchmarks is described in this section. The system used was the LANL ACL xed cluster, which is composed of three types of Compaq Alpha-based compute nodes. There are a total of 124 nodes, containing 152 processors with a total of 200GB of memory. The DS-10 and CS-20 nodes are interconnected using switched 100 megabit ethernet, while the ES-40 nodes are connected to the 100 megabit ethernet switch using gigabit ethernet. The specifications of the nodes are given in Table 1.

The benchmarks were run from the front end node, which is a single ES-40 used for starting jobs and controlling compute nodes. Jobs are issued to compute nodes and managed by BProc [1], which provides a single-system image with respect to processes. Each benchmark attempts to read Supermon data samples as many times per second as possible, slowly working from a small number of samples up until the duration of sampling takes longer than a single second. Each sample contains all possible data provided by Supermon to reflect the performance in the worst case. In general, applications will request only a subset of the data, and will be capable of higher sampling rates than those that we report. The approximate data sizes are shown in Table 2, with variation of about 5-10 bytes per sample depending on the data values being transported.

4.3 Kernel module performance

The first performance test involves measuring the maximum sampling rate for a program reading directly

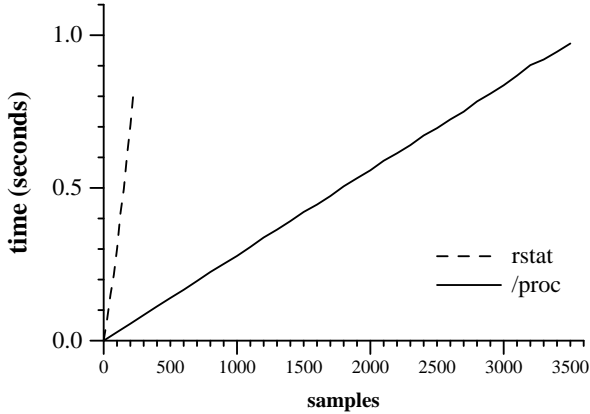


Figure 5. Number of samples vs time for `/proc` and `rstat`. `/proc` achieves a peak sampling rate of 3500 Hz while `rstat` can only achieve 300 Hz.

from the entries in `/proc` provided by the Supermon kernel module. We compare the performance of this portion of Supermon to the method used by RPC `rstat` to gather its data. The lowest sampling rates we found from `/proc` were 3400Hz on the DS-10 and CS-20 nodes, while the ES-40 nodes achieved 6000Hz. This test was also run on an Intel Pentium III machine, with performance comparable to the ES-40.

Comparing this to the performance of the `get_stats()` call used by `rstat` to gather performance, we find that we see a huge performance improvement (Figure 5). Using the same benchmark program used for measuring `/proc` with a minor change to call `get_stats()` instead of reading a file, we observe a peak performance of 300Hz. Not only is this an order of magnitude slower than `/proc`, later we will see that this is slower than the sampling rates observed after the data has passed through a single mon process and a single Supermon process.

4.4 Mon performance

To measure the performance of mon and Supermon, we used a similar program as used for measuring `/proc`. Instead of opening a file handle and reading, the benchmark opened a socket to the data server, sent a command asking for all data provided by the server, and read it before sending another request. In the case where mon is observed, the data must pass over two channels - first, it is read from the `/proc` entry and then sent between mon and the client over TCP.

4.5 Supermon performance

Using the same benchmark used for mon (since they use the same protocol), we measured the maximum sampling rate for various configurations of Supermon. The first is the case where Supermon is gathering data from a single mon process (Figure 6). This allows us to observe the effects of multiple hops across the network. For the case of multiple nodes, we tested with 5, 10, 20, and 100 nodes being monitored by a single Supermon process. Finally, we tested the performance of Supermon when they are constructed in a hierarchical topology using two cases: where each Supermon has a fanout of 10 nodes, and where each Supermon has a fanout of 50 nodes. At the root of the hierarchy a single Supermon process was used to gather the entire cluster data set from the Supermons observing subsets of nodes.

To make the test environment closer to one that would be encountered in practice, we took care to lay the Supermon servers out so that each Supermon was run on the first compute node in each subset of nodes. For example, if we were to monitor 100 nodes in groups of 10, a Supermon server would be run on node 0, 10, 20, etc. The Supermon server responsible for gathering data from each subset of nodes was run on a computer outside of the set of compute nodes, and the client was run on the cluster front end. This separation of Supermons was done for two reasons. First, we wanted to avoid overwhelming a single machine with many Supermon servers exchanging data. Second, we wanted to avoid any effects caused by loopback devices or TCP optimizations for socket communication within a single computer. This would generate results that would potentially show higher sampling rates, but it disregards the effect of the network on the monitoring process.

In Table 4.5, we show the performance results when testing the scalability of Supermon. The number of nodes corresponds with the number of mon servers a single Supermon connects to. In the case where all 100 DS-10 nodes were monitored, we provide three different cases to show the effect of hierarchical Supermon servers on performance. The basic case involves a flat topology where a single Supermon connects to 100 nodes. When composing Supermon servers into a tree topology, we test the cases where each Supermon connects to 10 clients (a single Supermon connected to 10 other Supermons, each of which is connected to 10 mons), and where a single Supermon connects to two Supermons responsible for monitoring half of the cluster each (50 mons). We were surprised to find that contrary to popular belief, hierarchy is not guaranteed to increase performance. In our case we showed that

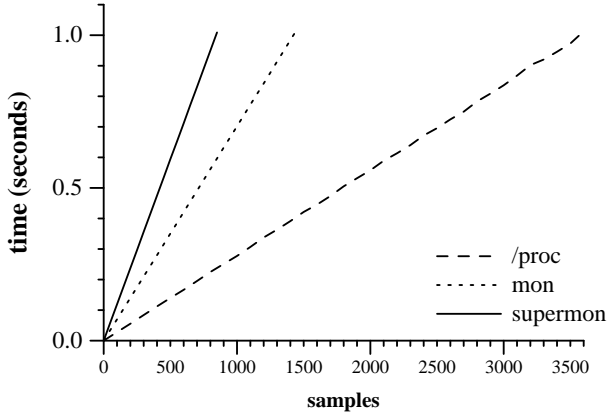


Figure 6. Number of samples vs time, comparing sampling rate from `/proc`, `mon`, and `Supermon`. Illustrates decrease in maximum sampling rate due to the network.

<i>Nodes</i>	<i>Sampling Rate</i>
5	400Hz
10	225Hz
20	125Hz
100 <i>Flat</i>	66Hz
100 <i>10-node fanout</i>	57Hz
100 <i>50-node fanout</i>	35Hz

Table 3. Scaling results for `Supermon`.

the additional network traffic between the layers of `Supermon` servers impacted negatively on the sampling rates achieved.

5 Conclusions and Future Work

`Supermon` is a set of tools for monitoring clusters at data rates heretofore impossible. Using `Supermon` we have been able to observe fine-grain behavior in systems built with MPI that has never before been seen. `Supermon` demonstrates the importance of having high data rate, low-impact cluster monitoring. The performance of the monitoring system is crucial. Formerly, the only tools available (e.g. `xload`, `xmeter`, and other `rpc.rstatd`-based tools) would have adverse impact on applications running on the cluster nodes at 10 Hz., and did not produce enough information to be usable.

We have shown that we can extract self-describing data from the kernel at a high data rate, up to 6000 samples/second on a Pentium 3/800 system. This data rate is as fast as using the `sysctl` interface [5], and over

100 times faster than the current generation of `/proc` interfaces. There need be no penalty for using `/proc` for gathering kernel statistics.

We have found that S-expressions are an ideal interface for getting information from the kernel. S-expressions have a structure that can handle kernel resources coming and going (e.g. PCMCIA cards), which no existing Linux interface can support. We feel that all existing Linux `/proc` interfaces should be redone to use S-expressions, replacing the current large number of inconsistent output formats.

`Supermon` also shows that RPC-based protocols such as SunRPC are neither necessary nor sufficient for this type of monitoring. They are not necessary as they have no performance advantage over the textual `Supermon` protocol. They are not sufficient as they do not handle variable-sized data well, and worse, require complex overhead at each end for converting data between architecture types. The textual S-expression format is inherently architecture-independent and just as efficient. It is long past time to retire SunRPC for this use and to eliminate the `rpc.rstatd` daemons as well.

Our next steps for `Supermon` are to further test scaling. We are looking at placing intermediate daemons (filtermons) in the tree to aggregate the data, so that a collection of, e.g., 32 nodes would be presented as an average. We will be putting more hardware information into `Supermon`, in addition to the limited network hardware statistics we have now. We will also be tying `Supermon` into our scheduler, so that the scheduler can make scheduling decisions based on hardware availability.

In addition to improving the monitoring system, we will also be looking at techniques for analyzing monitoring data for failure prediction, algorithm analysis, and performance optimization. In keeping with the spirit of the monitoring framework, the analysis techniques will not only be looked at from an analytical perspective but in terms of their computational complexity. This focuses on their ability to produce results using high-speed data samples at runtime instead of post-mortem analysis of stored monitoring data.

References

- [1] Erik Hendriks. BProc: The Beowulf Distributed Process Space. *Submitted for publication*, 2002.
- [2] Raj Jain. *The Art of Computer Systems Performance Analysis*, chapter 33, pages 563–567. John Wiley and Sons, Inc., 1991.

- [3] John McCarthy, et al. *LISP 1.5 Programmer's Manual, 2nd edition*. MIT Press, 1965.
- [4] Sean MacGuire. The Big Brother Unix Network Monitor. <http://www.csd.uwo.ca/bb/bb-info.html>.
- [5] Ronald Minnich and Karen Reid. Supermon: High performance monitoring for linux clusters. In *The Fifth Annual Linux Showcase and Conference*, 2001.
- [6] Sun Microsystems, Inc. XDR: External data representation standard. Network Working Group RFC 1014, <http://www.ietf.org/rfc/rfc1014.txt>, 1987.
- [7] Sun Microsystems, Inc. RPC: Remote procedure call protocol specification version 2. Network Working Group RFC 1057, <http://www.ietf.org/rfc/rfc1057.txt>, 1988.
- [8] TurboLinux. Powercockpit white paper. <http://www.powercockpit.com/markets/index.html>.